VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace kritické cesty
Visualization of a Critical Path

2014 Jan Koběluš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Koběluš**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vizualizace kritické cesty
Visualization of a Critical Path

Zásady pro vypracování:

Kaira je nástroj určený pro modelování a generování paralelních/distribuovaných aplikací. Aktuálně jsou generovány C++ aplikace využívající MPI. Hlavním cílem práce bude najít kritickou cestu v tracelogu aplikace. Cíle práce lze shrnout v těchto bodech.
1. Seznamte se s nástrojem Kaira a formátem v jakém jsou uložena data z profilování.
2. Navrhněte, jak z těchto dat získat kritickou cestu. Tedy cestu při vykonávání na které nedocházelo k čekání na další výpočty.
3. Prakticky implementujte navržené řešení. Realizujte vizualizaci nalezení kritické cesty.
4. Demonstrujte možnosti řešení na praktických příkladech.
K řešení využijte aktuálně používané technologie, specificky jazyky Python a C++.

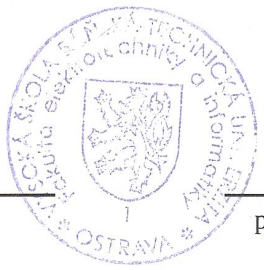
Seznam doporučené odborné literatury:

[1] S. Böhm, M. Běhálek: Usage of Petri nets for high performance computing, Functional high-performance computing, ser. FHPC '12. New York, NY, USA: ACM, 2012, pp. 37–48.
[2] Domovské stránky projektu Kaira - http://verif.cs.vsb.cz/kaira/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 18. července 2014 .....

Abstrakt

Tato práce se zabývá tvorbou rozšíření pro nástroj Kaira, umožňující nalezení a vizualizace kritické cesty z dat zaznamenaných při běhu programů vygenerovaných tímto nástrojem. Popisuje aplikaci metody kritické cesty na programy vygenerované Kairou, možné způsoby vizualizace a vypracování zvoleného řešení.

Klíčová slova: Kaira, kritická cesta, Petriho sítě, vizualizace

Abstract

This paper focuses on the developement of extension for the tool Kaira, enabling finding and visualization of a critical path from data collected during runtime of a program generated by this tool. It describes application of the Critical Path Method on programs generated by Kaira, possible ways of visualization and implementation of the chosen solution.

Keywords: Kaira, critical path, Petri nets, visualization

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Výpočetní technika je nedílnou součástí každodenního života a nároky na výkon a efektivitu se neustále zvyšují. To vede ke stále většímu využívání paralelních aplikací, které k výpočtům využívají více paralelně běžících procesů. Tyto paralelní systémy však sebou přinášejí řadu překážek a výzev. Vývoj se stává složitější a ke zlepšení efektivity je třeba implementace neustále vylepšovat. Důležitým předpokladem pro zlepšení implementace je zevrubná analýza. Metody analýzy se proto stále rozšiřují a umožňují lepší porozumění běhu aplikací.

Tato práce se má za cíl rozšířit analytické schopnosti nástroje Kaira o nalezení a vizualizaci kritické cesty z tracelogu vygenerovaného programu. Kapitola 2 obsahuje popis Petriho sítí a seznámení se s nástrojem Kaira. Kapitola 3 se zabývá problematikou kritické cesty, původním využitím, současným použitím v paralelním programování a její aplikací na programy vygenerované nástrojem Kaira. Kapitola 4 popisuje nalezení kritické cesty z tracelogu vygenerovaných programů, způsoby její vizualizace a předložení výsledků uživateli. Kapitola 5 nastiňuje možná budoucí zlepšení a rozšíření práce.

2 Kaira

Tato kapitola se zabývá úvodem do Petriho sítí a seznámením se s nástrojem Kaira.

2.1 Petriho sítě

Petriho síťe[1] jsou matematický nástroj pro popis paralelního chování a pro modelování distribuovaných systémů. Představují orientovaný graf tvořený přechody, místy a hranami, kdy přechody jsou zobrazeny jako obdélníky, místa jak kolečka a hrany šipkami. Hrany mohou být mezi místem a přechodem (vstupní hrana) nebo mezi přechodem a místem (výstupní hrana). Hrany nikdy nespojují dvě místa nebo dva přechody. V místech jsou umístěny anonymní entity označované jako tokeny. Podle počtu tokenů v jednotlivých místech můžeme určit stav sítě. Tokeny jsou zobrazovány jako černé tečky uvnitř míst. Manipulaci s tokeny obstarávají přechody. Místa můžeme rozdělit podle druhu hrany spojující je s přechodem na vstupní a výstupní. Příklad Petriho sítě je na obrázku 1. Při odpálení přechodu se odeberou tokeny ze vstupních míst a přidají se tokeny do míst výstupních 2. Přechod může být odpálen pouze tehdy, když je ve vstupních místech dostatek tokenů.

Formalně jsou Petriho sítě definovány[4] jako $PTN = (P, T, I, O, m_0)$ kde P je konečná množina *míst*, T je konečná množina *přechodů*, $P \cap T = \emptyset$, $I : T \times P \to N$, definuje mnohonásobnost vstupních hran a $O : T \times P \to N$ definuje mnohonásobnost výstupních hran, kde $N = \{0, 1, 2, ...\}$. *Značení* je mapování $P \to N; m_0 : P \to N$ je *původní značení*. Přechod t je povolen ve značení *m*, jestliže $\forall p \in P : m(p) \geq I(t, p)$. Jestliže je přechod povolen v *m*, pak může být odpálen a systém se dostane do nového stavu m', kde platí $\forall p \in P : m'(p) = m(p) - I(t, p) + O(t, p)$.

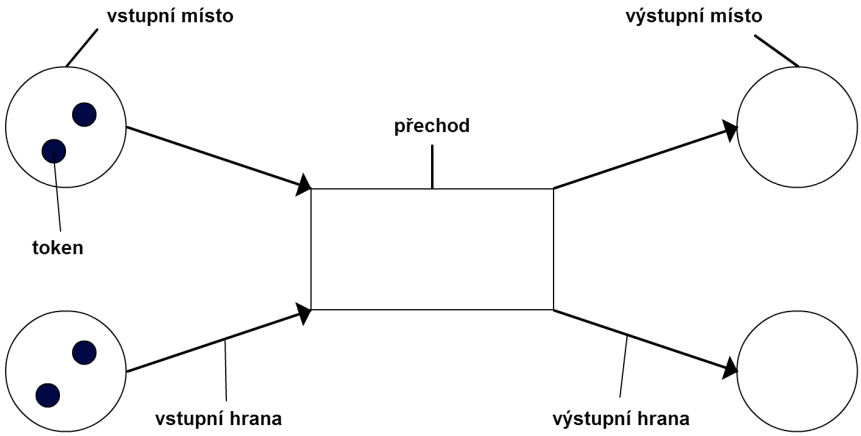
Barevné Petriho sítě (CPN)[2][3] jsou rozšířená verze Petriho sítí. Na rozdíl od základních Petriho sítí nepovažují barevné Petriho sítě tokeny za anonymní entity. Tokeny v CPN nesou hodnotu, takže stav sítě už neurčujeme pouze podle počtu tokenů v místech, ale podle hodnot tokenů v místech umístěných.

2.2 Kaira

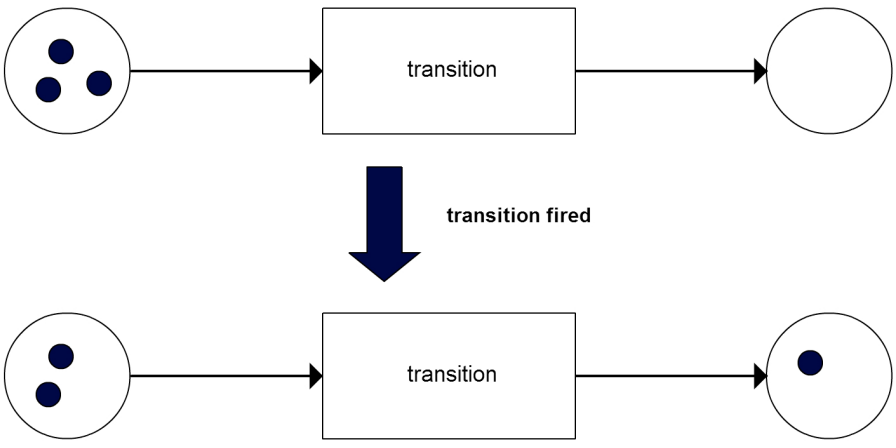
Kaira[1][4] je nástroj pro modelování paralelních aplikací vycházející z barevných Petriho sítí. Ukázka uživatelského rozhraní s otevřeným příkladem Workers na obrázku 3. Kaira mj. umožňuje:

- Tvorbu a editaci programu pomocí vizuálních prvků.
- Vkládání C++ kódu do vizuálních modelů
- Překlad vizualních modelů do C++ MPI (Message Passing Interface) programů a knihoven
- Ukázku a kontrolu programu ve vizuálním debuggeru

[1] http://verif.cs.vsb.cz/kaira/



Obrázek 1: Petriho síť

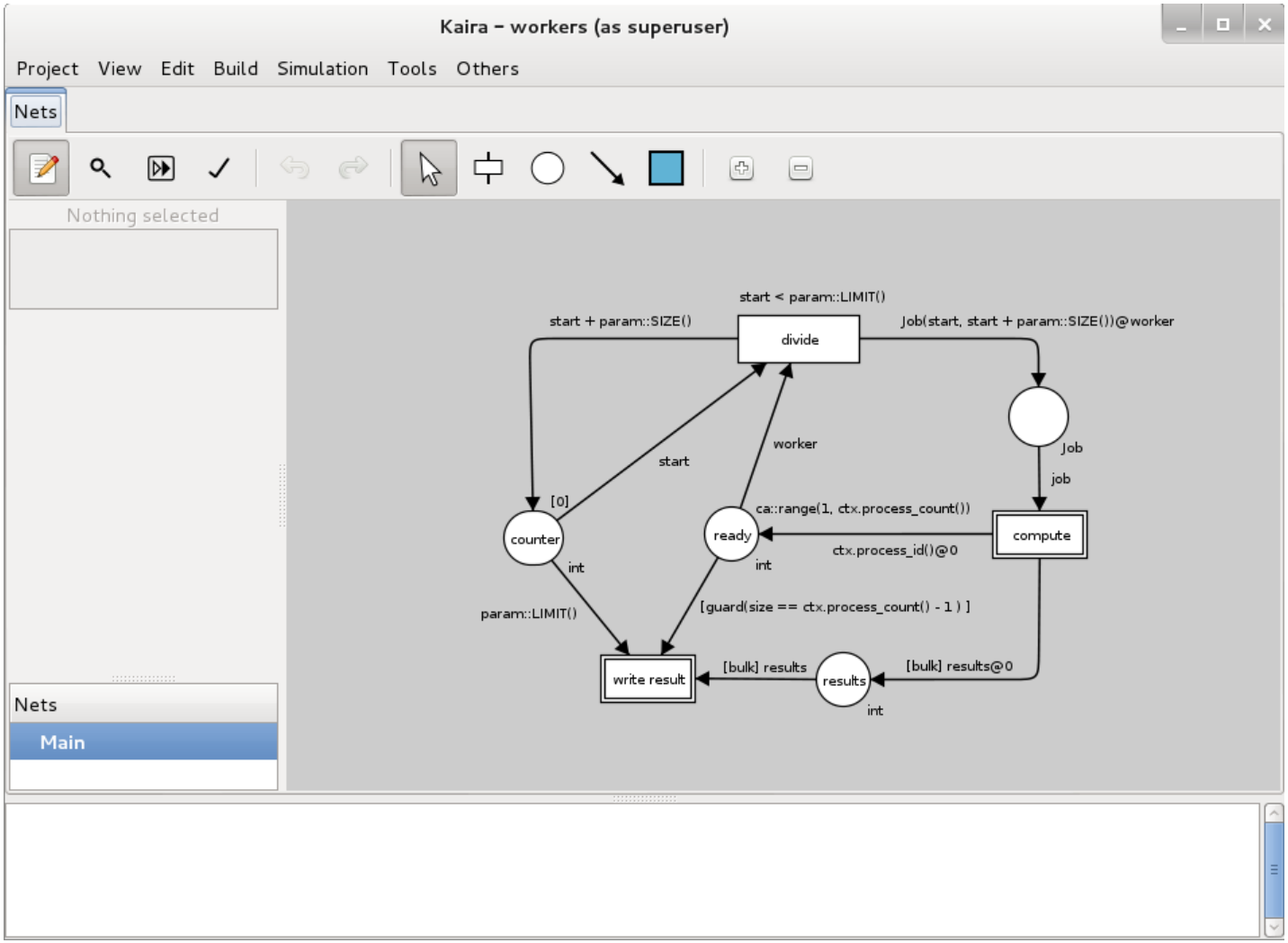


Obrázek 2: Petriho síť před a po odpálení přechodu

- Konfiguraci a vygenerovaní verze zachycující svůj průběh, s možností nahraní zaznamenaného průběhu programu zpět do Kairy

Důležitou součástí Kairy je simulátor. Ten umožňuje spuštění programu v prostředí plně kontrolovaném uživatelem. Umožňuje sledovat chování programu během všech kroků programu a poskytuje uživateli vizuální znázornění změn stavů programu.

Pro tento projekt nezbytnou součástí Kairy je funkce zaznamenávání průběhu programu. To je možné ve verzích sestavených v Trace módu. Před sestavením programu má uživatel možnost specifikovat, které údaje si přeje zaznamenávat. Při spuštění takto sestaveného programu je poté nutno použít parametr `-T SIZE`, kde `SIZE` představuje velikost paměťového bufferu v bytech. Výsledný tracelog se skládá z jednoho hlavičkového souboru a dále souboru pro každý proces využitý při běhu programu. Kaira umožňuje vytvořený tracelog zobrazit a analyzovat, popř. použít jako vstupní soubor některých rozšíření.



Obrázek 3: Kaira - uživatelské rozhraní s otevřeným projektem Workers

3 Kritická cesta

Tato kapitola se zabývá problematikou kritické cesty, obecnou metodou k jejímu nalezení, využitím kritické cesty v paralelním programování a popisuje teoretický základ pro implementaci nalezení kritické cesty v rámci nástroje Kaira.

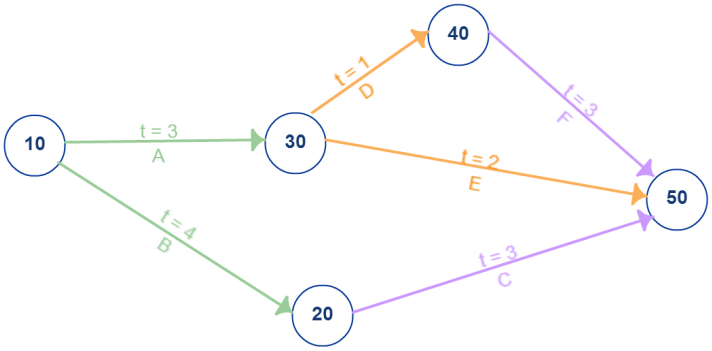
3.1 Úvod do kritické cesty

Metoda kritické cesty (CPM [5] je algoritmus používaný při plánování projektů sestávajících z množství aktivit. Používá se zejména pro zajištění efektivního řízení projektu. Vyvinut byl v padesátých letech 20. století v rámci spolupráce firem DuPont (E. I. du Pont de Nemours and Company) a Remington Rand. Autory jsou Morgan R. Walker z Dupontu a James E. Kelley, Jr. Z Remington Rand.

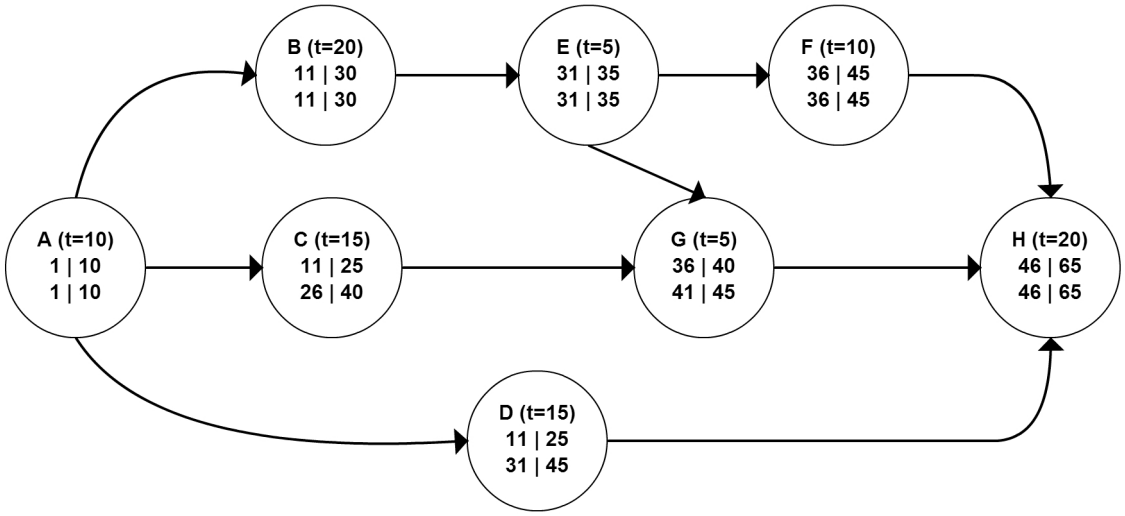
K nalezení kritické cesty je potřeba vytvořit model projektu ve formě acyklického orientovaného grafu, který obsahuje jednotlivé dílčí úkoly projektu, čas potřebný k jejich vykonaní a závislosti mezi jednotlivými úkoly. Vytvořený model může mít dvě podoby lišící se zobrazením úkolů. První model zobrazuje úkoly jako šipky mezi uzly představující stavy projektu, příklad na obrázku 4, zatímco druhý model zobrazuje úkoly jako uzly, které jsou navzájem propojeny šipkami představující závislosti, příklad na obrázku 5. V rámci této práce se bude nadále pracovat pouze s modelem obsahujícím aktivity jako uzly.

Cesta je označení průchodu grafem z výchozího uzlu do koncového uzlu, kdy je možno mezi uzly přecházet jen ve směru závislosti. Kritické cesta je označení takové cesty, u které prodloužení doby trvaní libovolného úkolu o libovolnou časovou hodnotu vyústí v prodloužení doby trvání celého projektu. Kritická cesta tedy představuje časově nejnáročnější průchod modelem projektu. Projekt může obsahovat více kritických cest. Ostatní cesty jsou označovány jako cesty nekritické a u jejich prvku lze zjistit tzv. float (slack), což je časová hodnota o kterou lze prodloužit dobu trvání úkolu bez ovlivnění doby trvání celého projektu.

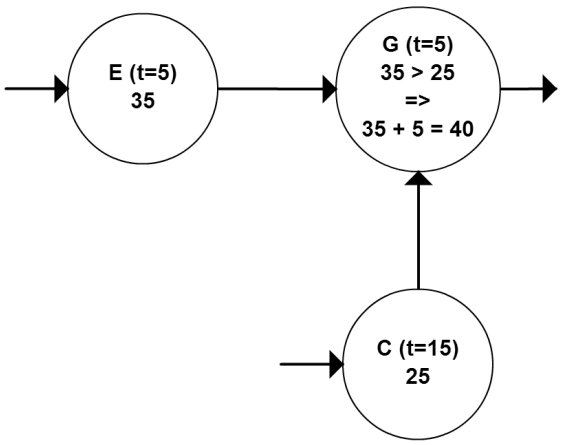
Při vyhledávání kritické cesty se nejprve orientovaným grafem prochází od počátečního uzlu po směru orientovaných hran. Zapisuje se hodnota ukončení jednotlivých úkolů získána přičtením doby, kterou daný úkol potřebuje k vykonání, k nejvyšší hodnotě u úkolů, na kterých je daný úkol závislý, viz obrázek 6. Po zjištění těchto hodnot u všech uzlů grafu se provede druhý průchod grafem, tentokráte od koncového uzlu proti směru orientovaných hran. Při tomto průchodu postupně odečítáme dobu trvání od hodnoty posledního úkolu získané v prvním průchodu a mezi úkoly po hranách přecházíme tak, aby hodnoty zapisované do jednotlivých uzlů byly co nejmenší, příklad na obrázku 7. Po provedení druhého průchodu grafem porovnáme získané hodnoty. Jestliže se hodnota uzlu získaná v prvním průchodu rovná hodnotě získané v průchodu druhém, pak daný uzel leží na kritické cestě.



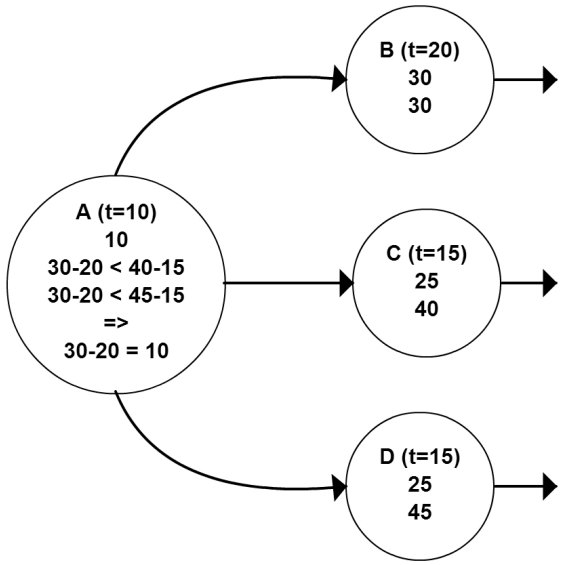
Obrázek 4: Diagram s aktivitami na šipkách



Obrázek 5: Diagram s aktivitami na uzlech



Obrázek 6: Výpočet hodnoty aktivity při prvním průchodu orientovaným grafem



Obrázek 7: Výpočet hodnoty aktivity při druhém průchodu orientovaným grafem

3.2 Kritická cesta v paralelním programování

Přestože CPM byla původně vyvinuta pro plánování a koordinaci komplexních inženýrských projektu, myšlenka kritické cesty má využití také v analýze výkonu paralelních programů. Mezi nástroje využívající kritickou cestu patří např. Scalasca.

Scalasca[2][7] je softwarový nástroj měřící a analyzující chování paralelních programů během jejich běhu. Tato analýza identifikuje potenciální výkonové překážky (bottlenecks), zejména ty v rámci komunikace a synchronizace, a umožňuje jejich hlubši prozkoumání. Scalasca je určena hlavně pro aplikace využívající rozhraní MPI a OpenMP, včetně hybridních aplikací využívající kombinaci obou. Nástroj byl vytvořen pro large-scale systémy jako jsou IBM Blue Gene a Cray XT, ale lze použít i pro small-scale a medium-scale HPC (High performance computing) platformy.

Pro vyhodnocení chování paralelních programu zachytává Scalasca data během běhu programu, která jsou určená pro vyhodnocení po ukončení programu. Uživatel si muže vybrat mezi dvěma způsoby analýzy: Přehled výkonu pomocí profilování nebo Studii chování aplikace pomocí sledování událostí. V profilovací části generuje Scalasca celkové výkonové metriky pro jednotlivé funkční cesty, což je užitečné k identifikaci částí nejnáročnějších na prostředky. Při sledování události jsou zaznamenávány jednotlivé události, což umožňuje automatickou identifikaci cest a wait-states. Reporty vytvořené oběma způsoby je možno procházet v interaktivním grafickém prohlížeči.

Rozšíření nástroje Scalasca o analýzu využívající CPM[8] je určeno zejména pro multiple program multiple data (MPMD), ale je využitelné i pro single program multiple data (SPMD). Využívá kombinaci kritické cesty a profilů pro jednotlivé procesy k odvození výkonových indikátorů. Tato analýza poskytuje dvě skupiny výkonových dat.

První skupina, profil kritické cesty a ukazatel nevyváženosti kritické cesty, popisuje dopad dílčích částí programu na dobu vykonání. Profil kritické cesty reprezentuje čas potřebný pro aktivity kritické cesty. Indikátor nevyváženosti ukazuje množství času ztraceno kvůli nevyváženosti zátěže jednotlivých cest. Tyto dvě metriky poskytují přehled cílů pro optimalizaci a paralelní neefektivnosti v SPMD programech.

Druha skupina, určena hlavně pro MPMD programy, se zabývá vlivem dílčích úkolu programu na využití dostupných prostředků. Dokáže naleznout zdroj nevyváženosti a rozlišit, jestli je neefektivní využití prostředků způsobeno nerovnoměrným rozdělením nebo nevyvážeností mezi procesy vykonávající stejnou aktivitu v rámci jedné části.

Kritická cesta je nalezena pomocí zpětného průchodu zachycených událostí a pomocí wait-state analýzy Scalascy postupně nalezne předchozí prvky kritické cesty. Po nalezení celé kritické cesty odvozeny jednotlivé výkonové indikátory, které jsou následně vyhodnoceny v rámci nástroje Scalasca.

K ověření efektivity této analýzy použili autoři aplikaci simulující různé nevyváženosti. Aplikace vykonávala funkci ve smyčkách po nastavitelnou dobu a synchronizovala procesy na konci jednotlivých cyklů. Byly použity čtyři různé případy.využívající 32 procesu. Celkové využití procesů bylo shodné, lišila se ale distribuce zátěže na jednotlivé procesy. V prvním případu byla zátěž vybalancována shodně mezi všechny procesy. Ve

[2] http://scalasca.org

zbylých třech případech byly úměle vytvořeny nevyváženosti prodlužující dobu potřebnou k vykonaní o 25%.

Každý z těchto případů byl spuštěn jako 320 iterací cyklu. Vybalancovaný příklad byl dokončen za 16.15 vteřiny, příklady s úmělou nevyvážeností trvaly mezi 19.98 a 20.04 vteřinami. Analýza využívající kritickou cestu nalezla mezi 3.87 a 3.99 vteřinami nevyváženosti u každého nevyváženého příkladu a tyto hodnoty se velmi blíží k nastavenému prodloužení o 25%. V porovnaní s analýzou využívající profilování procesů vyšla analýza využívající kritickou cestu jako stejně efektivní v příkladu se statickou nevyvážeností. V příkladech kde se objevovala nevyváženost dynamická byla analýza využívající kritickou cestu výrazně lepší.

3.3 Kritická cesta v rámci Kairy

Při vytváření modelu programu[9] vytvořeného pomocí nástroje Kaira se vychází z modelu používaného při obecné metodě kritické cesty. Přechody odpálené během průběhu programu jsou zobrazený jako uzly, které jsou propojeny šipkami představující závislosti mezi přechody. Na rozdíl od modelu CPM může tento model obsahovat více počátečních a koncových stavů. Příklad modelu je k vidění na obrázku 8. Osa X představuje časový průběh programu a na ose Y jsou zobrazeny procesy programem využité. Na ukazkovém diagramu je k program sestávající z jedenácti přechodu využívající čtyři procesy. Přechod A je přechodem výchozím a do míst přidává tokeny potřebné k odpálení přechodu B, C, D a E. Přechod B přidává token potřebný pro odpálení přechodů F, který následně přidává token potřebný k odpalení přechodu I. Přechod C přidává tokeny pro přechody G a H, kdy přechod H probíhá na stejném procesu jako přechod D. Přechod J k odpálení potřebuje tokeny dodané přechody I a G. Koncový přechod K k odpalení vyžaduje tokeny od přechodů J a H a také uvolnění procesu od přechodu E.

Ke správnému vytvoření modelu je u jednotlivých přechodů třeba znát:

- ID
- čas odpálení
- čas dokončení
- proces
- vstupní a výstupní místa
- odebrané a přidané tokeny

Tyto informace jsou využity ke správnému umístění přechodu v modelu a k nalezení závislostí mezi přechody.

Přechod B je na přechodu A závislý právě tehdy, když přechod B nemohl být odpálen bez předchozího dokončení přechodu A. Závislosti lze rozdělit na dva druhy:

1. Výpočetní: v jednom okamžiku může na jednom procesu probíhat pouze jeden přechod. Přechod je tedy výpočetně závislý na všech předchozích přechodech, které

proběhly na stejném procesu. To je možno zjednodušit pouze na závislost na posledním proběhnuvším přechodu na daném procesu.

2. Datová: Odstraňuje-li přechod B tokeny z místa, do kterého byly dané tokeny přidány přechodem A, je přechod B datově závislý na přechodu A.

Přechod může být závislý na libovolném množství přechodů. Přechod může být závislý na stejném přechodu výpočetně i datové. V konečném modelu nejsou zobrazeny všechny nalezené závislosti, pro přehlednost jsou odstraněny závislosti odvoditelné pomocí pravidla tranzitivity.

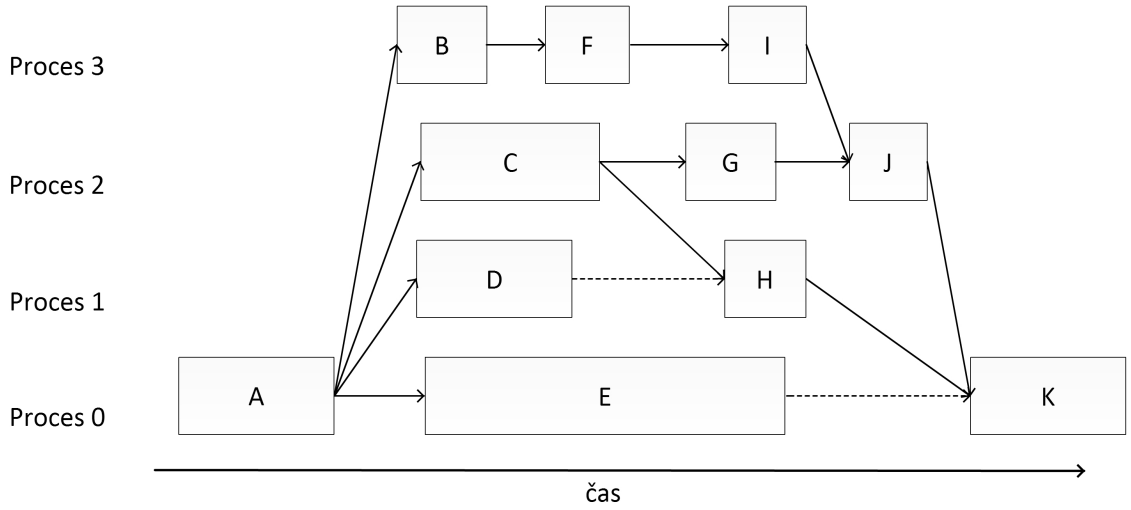
Ve správně vytvořeném modelu jde vyhledat kritická cesta. Na rozdíl od obecné CPM se využije pouze jeden průchod grafem, který začne u posledního dokončeného přechodu. Tento přechod je posledním prvek kritické cesty. Předcházející prvek kritické cesty je přechod z množiny přechodů, na kterých je poslední prvek závislý, a byl dokončen jako poslední. Ukázka výběru správné závislosti na obrázku 9. Takto se postupuje, dokud se grafem nepřejde na přechod, který není závislý na žádném jiném přechodu. Tento přechod je prvním prvek kritické cesty. Teoreticky je možné, že přechod bude závislý na více přechodem končících ve stejném čase. Pokud tato vysoce nepravděpodobná situace nastane, vybere se závislost, která byla odpálena později.

Takto vytvořený model průběhu programu s nalezenou kritickou cestou lze použít k další analýze programu a nalezení problémů při jeho běhu. Mezi možné problémy patří:

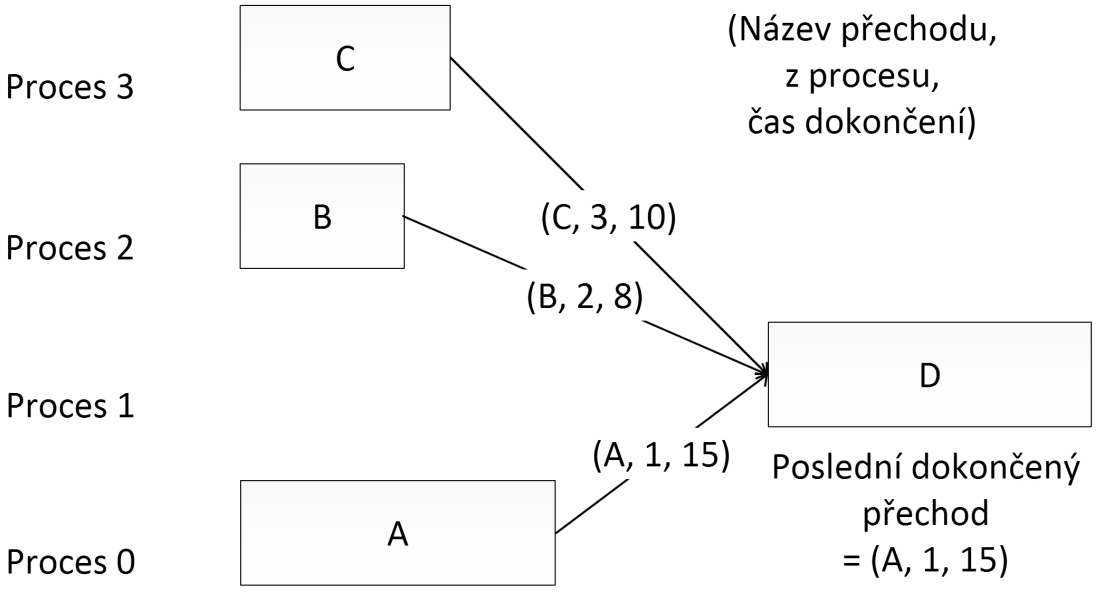
- Velké rozdíly mezi kritickou cestou a cestami nekritickými. V ukázce na obrázku 10 je příklad s kritickou cestou skládající se z prechodů A, B, F, G a H a třemi nekritickými cestami. Tyto nekritické cesty mají stejný výchozí i konečný přechod s cestou kritickou, ale část kterou se od kritické cesty liší je vykonána podstatně rychleji než přechody cesty kritické, což způsobuje nevyužití tří ze čtyř dostupných procesů po vetšinu doby potřebné k dokončení programu

- Nedostatečné využití dostupných prostředků: Program dostatečně nevyužívá všechny dostupné procesy a tím je zpomalen jeho průběh. V příkladu na obrázku 11 lze ukázkový diagram rozdělit na dvě části. V části mezi přechody E a I jsou využity všechny dostupné procesy, zatímco v části mezi přechody A a E je využita pouze polovina dostupných procesu a tím je prodloužena celková doba trvání programu.

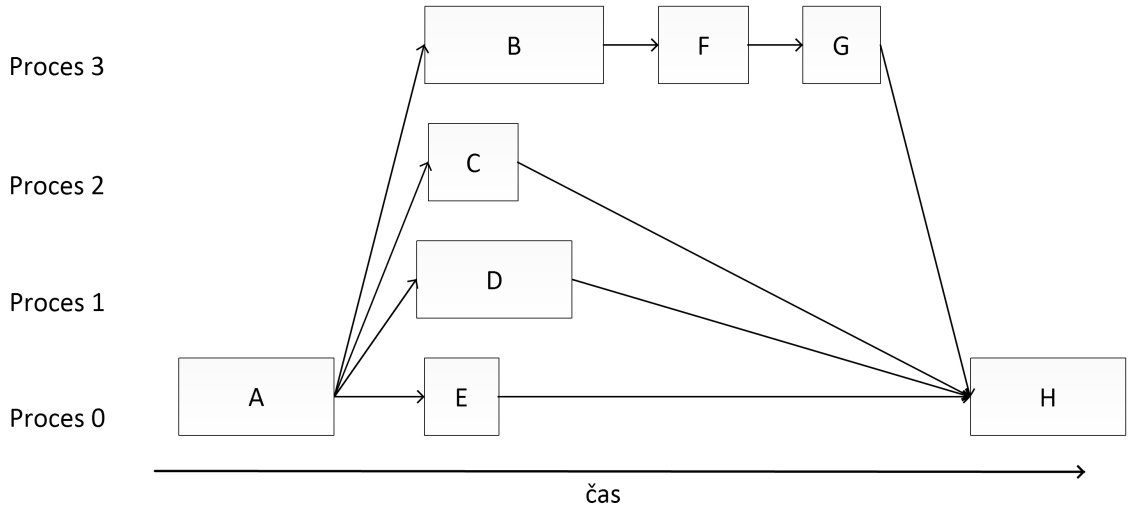
Kromě problémů lze v diagramech vypozorovat např. opakování stejných sekvencí přechodů nebo jestli jsou určité přechody vykonávány pouze na určitých procesech. Ze správně vytvořeného diagramu by uživatel měl získat základní představu o efektivitě programu a na které části se zaměřit při případné optimalizaci.



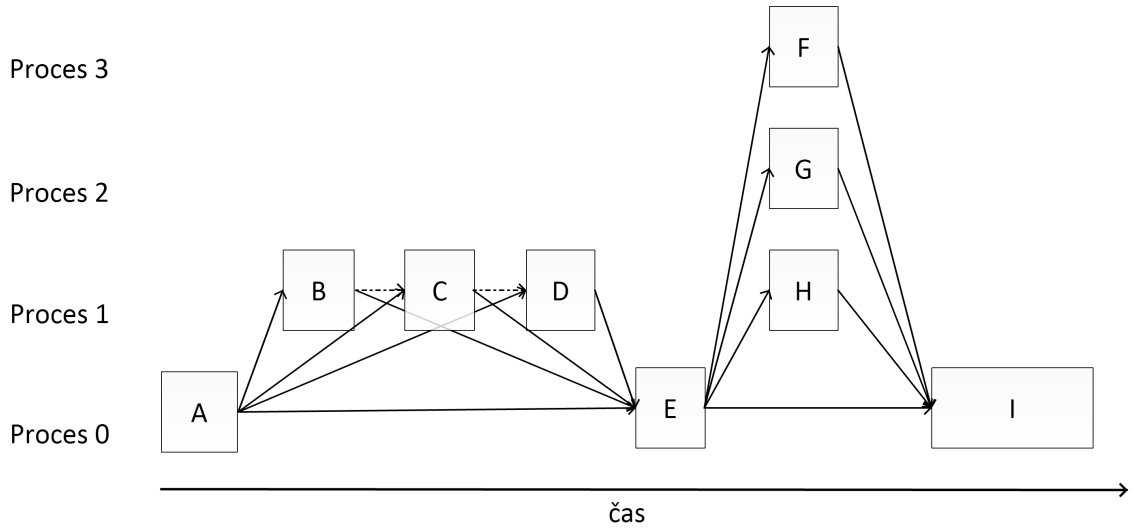
Obrázek 8: Ukázka diagramu znázorňujícího průběh programu



Obrázek 9: Výběr relevantní závislosti



Obrázek 10: Diagram znázorňující velký rozdíl mezi kritickou cestou a cestou nekritickou



Obrázek 11: Diagram znázorňující neefektivní využití dostupných procesů

4 Implementace

V této kapitole je popsáno řešení a implementace nalezení a vizualizace kritické cesty z dat zaznamenaných při běhu programu vygenerovaného nástrojem Kaira.

4.1 Myšlenka a požadavky

Pro nalezení kritické cesty je potřeba mít k dispozici data o průběhu programu. Je potřeba aby byl program v Kaiře sestaven v traced verzi a bylo nastaveno zachytávání informací o všech přechodech a tokenech, příklad na obrázku 12. Při spuštění takto vytvořeného programu je třeba zadat parametry pro vytvoření tracelogu, který bude obsahovat informace vyžadované pro nalezení kritické cesty. Potřeba jsou zejména informace o proběhlých přechodech, o místech a o pohybu tokenů. Z načtených dat jsou mezi jednotlivými přechody vypočteny závislosti. Tyto závislosti jsou následně zredukovány tak, aby se mezi nimi nevyskytovaly závislosti odvoditelné pomocí jiné závislosti. Poté se ve vzniklé síti přechodů najde kritická cesta, která je následně poskytnuta uživateli ve vizuální formě.

Zadání práce bude implementováno jako rozšíření programu Kaira a napsáno bude v jazyce Python.

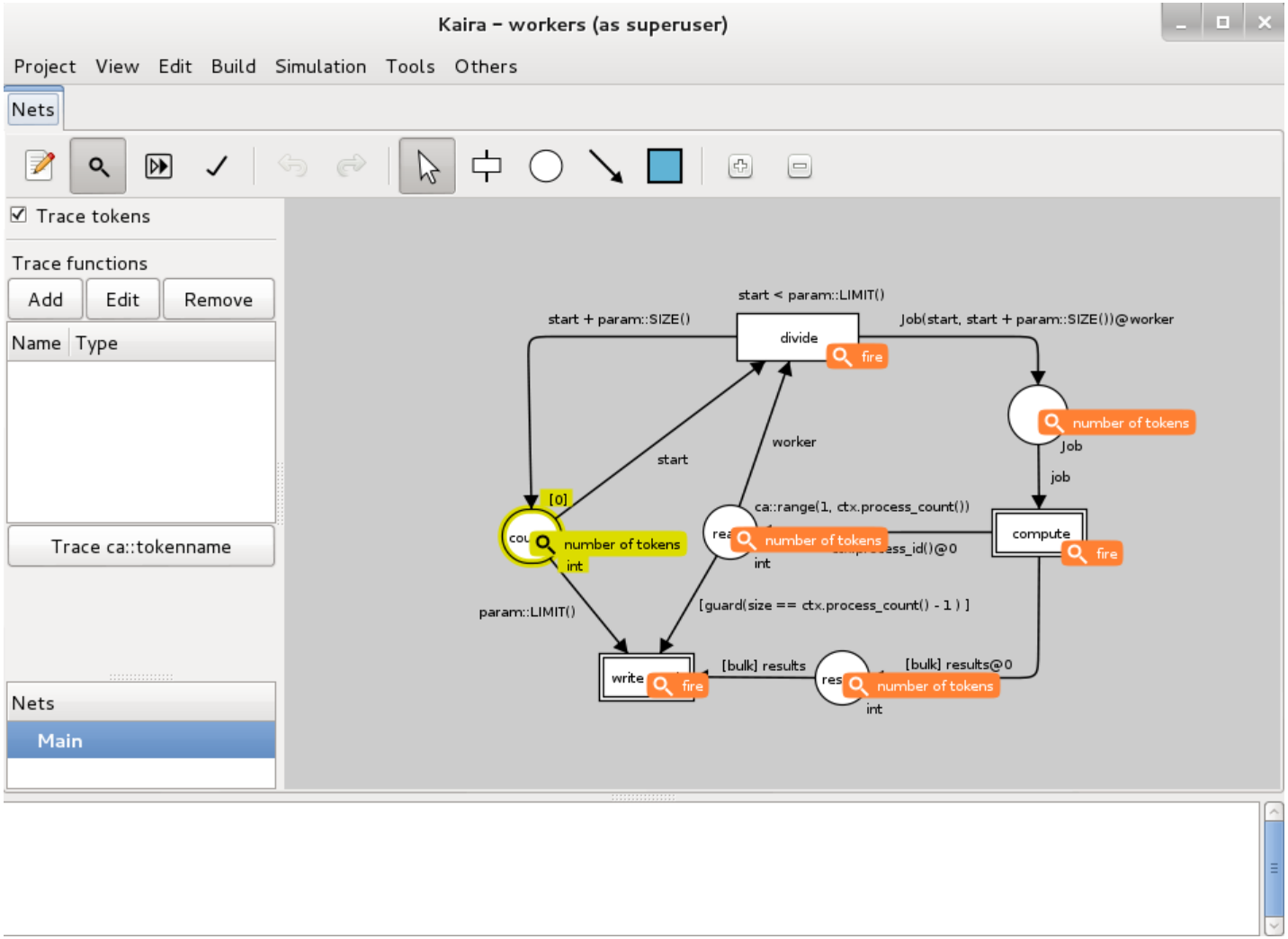
4.2 Základní informace

Vypracované rozšíření obsahuje tyto třídy:

- `CriticalPath`
- `ExportRunInstance`
- `TransitionNode`
- `EventTracker`
- `TransitionInfo`
- `GraphAddons`

Třída `CriticalPath` je hlavní třídou rozšíření a obsahuje metodu run zajišťující spuštění. Je potomkem `extensions.Operation`. Třída `ExportRunInstance`, potomek třídy `RunInstance`, má na starost načtení potřebných informací, jejich zpracování a vytvoření vizualizace. Instance třídy `TransitionNode` představují všechny proběhlé přechody, instance třídy `EventTracker` uchovávají informace o událostech vyvolaných přechody a instance třídy `TransitionInfo` uchovávají vybrané vlastnosti přechodů sítě projektu. Třída `GraphAddons` obsahuje rozšířené ovládací prvky používané při zobrazení výsledku a zprostředkovává předání informací o jednotlivých přechodech uživateli.

Rozšíření importuje následující části nástroje Kaira:



Obrázek 12: Kaira - nastavení pro traced build

- `extensions`
- `RunInstance` z `runinstance`
- `settingswindow`
- `RESPONSE_APPLY` z `gtk`
- `utils`
- `table`

Pro vykreslení grafu jsou použity balíčky

- `matplotlib.pyplot`
- `networkx`

`Networkx`[3] není jako jediná součást nainstalována v rámci standardní instalace Kairy.

4.3 Nalezení kritické cesty

4.3.1 Načtení dat z tracelogu

Pro nalezení kritické cesty jsou potřeba informace o všech proběhlých přechodech, tedy čas odpálení přechodů, jak dlouho a na jakém procesu probíhaly, a s jakými tokeny pracovaly. Pro získání těchto dat byla vytvořená třída `ExportRunInstance`. Při inicializaci instance třídy `ExportRunInstance` je vstupním parametrem `tracelog` a jsou během ní vytvořeny dva slovníky s informacemi o přechodech a místech vyskytujících se v programu. U slovníku s přechody jsou klíči id jednotlivých přechodů a jako hodnoty jsou instance třídy `TransitionInfo`, shrnující informace o přechodech – id, název, mezi kterými místy proces je – potřebných k nalezení kritické cesty. Ve druhém slovníku jsou klíči id míst a hodnotami jsou jejich instance.

Data jsou získávána pomocí rozšíření metod rodičovské třídy pracujících s relevantními údaji. Mezi tyto metody patří:

- `transition_fired`
- `transition_finished`
- `add_token`
- `remove_token`
- `event_send`
- `event_receive`

[3] https://networkx.github.io/

Během `transition_fired` dochází k vytvoření instance třídy `TransitionNode` s údaji o čase započetí přechodu, s číslem procesu, na kterém proces proběhl, s pořadovým číslem přechodu a odkazem na instanci třídy `TransitionInfo` získané ze slovníku vytvořeného pří inicializaci třídy `ExportRunInstance`. Délka přechodu je do instance `TransitionNode` přidána během `transition_finished`, kdy je vypočtena jako rozdíl času ukončení a začátku přechodu. Při metodě `add_token` (respektive `remove_token`) se do do listu `added_tokens` (`removed_tokens`) příslušné instance třídy `TransitionNode` přidají hodnoty `token_pointer`. Metoda `event_send` přidává do listu událostí příslušného `TransitionNode` novou instanci `EventTracker` s dvojicí tvořenou číslem procesu, na kterém daná událost vznikla, a číslem procesu, na který míří. Informace o přesunutých tokenech a čase dokončení se přidají během průběhu `event_receive`. Proběhnutí všech událostí zajišťuje metoda `execute_all_events` třídy `tracelog`. Výsledkem je list `transtion_list`, který obsahuje všechny instance `TransitionNode` s daty potřebnými pro nalezení závislostí a samotné kritické cesty.

4.3.2 Určení závislosti mezi načtenými přechody

Určení závislostí mezi načtenými přechody má na starost metoda `find_dependencies` třídy `TransitionNode`. Vstupem metody je seznam všech načtených přechodů. Metoda má dvě části:

- Vyhledání závislosti datové
- Vyhledání závislosti výpočetní

Při vyhledávání závislosti datové se zjišťuje, které přechody pracovaly s tokeny z listu `tokens_removed` dané instance `TransitionNode` . Listem všech přechodů se prochází od konce a kontrolují se pouze přechody, které proběhly dříve, než přechod pro který se závislosti vyhledávají. První se kontrolují tokeny přidáné v rámci všech událostí a poté se zkontrolují přechody v listu `tokens_added`. Při nalezení zdrojů jednotlivých tokenů z listu `tokens_removed` si instance `TransitionNode` poznačí závislost. Listem přechodů se prochází od konce, protože jednotlivé `token_pointer` se vyskytují v průběhu programu vícekrát, a tímto zajistíme, že se do závislostí bude započítávat pouze přechod, který s daným tokenem pracoval jako poslední a zabrání se tak vytváření zbytečných vazeb. Po zjištění všech datových závislostí se zjišťuje závislost výpočetní, kdy se do listu závislostí přidá poslední přechod, který probíhal na stejném procesu.

```
def find_dependencies(self,transitions):
    for token in self.tokens_removed:
        tokenFound = 0
        for t in reversed(transitions):
            all_added_tokens = []
            for e in t.tracked_events:
                if self.process_id == e.start[1] \
                    and self.position_in_list > t.position_in_list and token in e.tokens and
                        tokenFound == 0:
                    self.data_dependant_on.append(t.position_in_list)
                    tokenFound = 1
```

```
if self.position_in_list > t.position_in_list and token in t.tokens_added and
    tokenFound == 0:
  self.data_dependant_on.append(t.position_in_list)
  tokenFound = 1
```

Výpis 1: Nalezení datových závislostí mezi přechody

Nalezené závislosti je poté nutno zredukovat, ne všechny jsou totiž potřebné pro další průběh programu. Cílem redukce je, aby se jednotlivé závislosti mezi přechody nedaly odvodit pomocí závislostí jiných. K tomuto účelu obsahuje třída `TransitionNode` metodu `clear_dependencies`, která jako parametr vyžaduje seznam všech přechodů. Postup při vyčistění závislostí je následovný: pomocí rekurze se vytvoří množina závislostí všech přechodů v seznamu závislostí. Po vytvoření tohoto seznamu se vypočte rozdíl původního seznamu závislostí a nově vytvořeného seznamu obsahující množinu všech závislostí podmínek. Tento rozdíl je hledaná vyčištěná množina.

4.3.3 Vlastni nalezeni kritické cesty

Po vytvoření sítě závislostí mezi přechody už nic nebrání nalezení kritické cesty. Prvním krokem je zkonsolidování datových a výpočetních závislostí do jednoho listu, který je poté seřazen. Samotné vyhledávání začíná u posledního proběhlého přechodu, který je na kritické cestě vždy. Z něj přejdeme po hraně závislosti do přechodu, který poskytl potřebné tokeny (popř. uvolnil proces) jako poslední a označí se jako přechod patřící na kritickou cestu.

```
def find_critical_path(self):
  temp=0
  for t in self.transitions_list:
    if t.process_dependant_on!=None and t.process_dependant_on not in t.
        data_dependant_on:
      t.data_dependant_on.append(t.process_dependant_on)
      t.data_dependant_on.sort()
    if t.start_time+t.tet>self.transitions_list[temp].start_time+self.transitions_list[temp].tet
        :
      temp=t.position_in_list
    while temp>0:
      self.transitions_list[temp].on_critical_path=True
      if self.transitions_list[temp].data_dependant_on==[]:
        break
      temp2=self.transitions_list[temp].data_dependant_on[0]
      self.transitions_list[temp].on_critical_path=True
      for d in self.transitions_list[temp].data_dependant_on:
        if self.transitions_list[d].start_time+self.transitions_list[d].tet>=self.
            transitions_list[temp2].start_time+self.transitions_list[temp2].tet:
          temp=self.transitions_list[d].position_in_list
          temp2=self.transitions_list[d].position_in_list
    for t in self.transition_list:
      t.clear_dependencies(self.transition_list)
```

Výpis 2: Nalezení kritické cesty

Nalezená kritická cesta je uložena jako seznam `TransitionNode` v atributu `critical_path` třídy `ExportRunInstance`.

4.4 Vizualizace

Následující část práce se zabývá vizualizací nalezené kritické cesty. Pro demonstraci výsledků byly použity tyto příklady:

- workers
  `./workers -r4 -pLIMIT=120 -pSIZE=10 -T1M`
- sieve
  `./sieve -r4 -T1M -pN=100`

4.4.1 Možnosti vizualizace

K vizualizaci nalezené kritické cesty se nabízelo velké množství způsobů. Nad ostatní vyčnívaly tyto tři varianty:

- Zobrazení pomocí prvků pro tvorbu sítí v Kaiře
- Zobrazení jako interaktivní mapa
- Zobrazení pomocí orientovaného grafu

Zobrazení pomocí již existujících grafických prvků pro přechody a místa v Kaiře se nabízelo jako první řešení. Největší výhodou této možnosti zobrazení by byla obeznámenost uživatele s grafickými prvky. Zobrazena by byla rozvitá síť programu, se všemi proběhlými přechody a místy mezi nimi. Ačkoliv tato možnost může na první pohled působit ideálně, byla nakonec zamítnuta. Důvodů odklonění se od této varianty bylo více. Mezi hlavní patřila složitost vytvoření algoritmu pro vykreslení rozvité sítě programu tak, aby i u složitějších programů s velkým množstvím přechodů a míst byla výsledná mapa přehledná, jednotlivé prvky se nepřekrývaly a vyznačená kritická cesta byla jasně zřetelná. Výsledná sít by také byla ve většině případů mnohonásobně větší než sítě projektů, pro které je uživatelské rozhraní Kairy uzpůsobeno.

Zobrazení jako interaktivní mapa průběhu programu by byla v mnoha ohledech ideální varianta vizualizace. Výsledkem by byla interaktivní síť, kde by jednotlivé prvky představovaly přechody a obsahovaly by veškeré informace získané z tracelogu, tedy informace o přesunutých tokenech a mezi kterými místy přechod proběhl, čísle procesu, dále o času zahájení přechodu, času ukončení a době trvání. Závislosti mezi přechody by byly odlišeny podle druhu a kritická cesta by byla zvýrazněna odlišnou barvou. Uživatel by měl možnost zobrazené informace a přechody filtrovat. Toto řešení bylo zamítnuto z důvodů časové náročnosti.

Zobrazení pomocí orientovaného grafu se nabízí přímo z podstaty kritické cesty. Tato varianta umožňuje zobrazit všechny přechody a závislosti mezi nimi jako přehledný celek s možností jednoduchého zvýraznění kritické cesty. Tento způsob je v základu podobný variantě zobrazení pomocí již existujících grafických prvků Kairy, kdy nepřítomnost již známých grafických zobrazení pro přechody a místa vynahrazuje jednodušší

tvorbou a lepšími možnostmi zobrazení a prohlížení, a to zejména u složitějších programů.

Pro konečné zobrazení byla nakonec zvolená varianta zobrazení ve formě orientovaného grafu ve dvou variantách, doplněná o možnost textového zpracování pomocí tabulek.

4.4.2 Implementovaná vizualizace

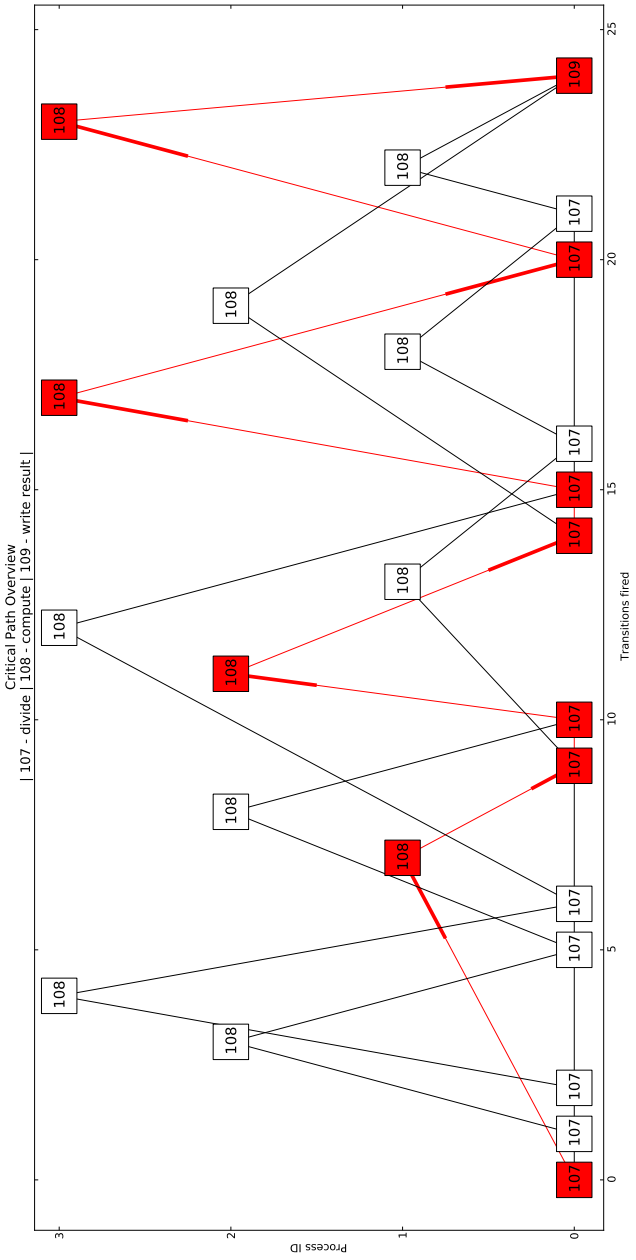
Výsledek rozšíření programu Kaira pro vyhledání kritické cesty je uživateli předán ve formě orientovaného grafu vytvořeného pomocí třídy `DiGraph` z balíčku `networkx` zobrazeného jako interaktivní obrázek balíčku `pyplot` z `matplotlib` API. Interaktivní obrázek byl jako způsob zobrazení zvolen namísto obrázku ve formě png nebo pdf hlavně díky možnosti úprav. Vzhled vygenerovaného grafu je závislý od počtu proběhlých přechodů a může se tedy výrazně lišit jak mezi různými programy, tak také mezí rozdílnými průběhy programu stejného. Proto je ukládání do obrázku podle předem nastavených parametru nepraktické. Interaktivní zobrazení umožňuje uživateli si výsledný graf přibližovat a oddalovat, zvětšovat čí zmenšovat velikost grafu nebo se zaměřit na určitou část a poté si uložit výsledek nastavený podle svých představ. Uživatel má možnost si vybrat ze dvou variant zobrazení, kdy první ukazuje celkový přehled průběhu programu jako sekvence propojených uzlů představujících jednotlivé přechody, zatímco druhá možnost je zaměřena na detailnější zobrazení kritické cesty a kromě přechodů obsahuje i relevantní místa.

4.4.2.1 Celkové zobrazení (Overview) V celkovém zobrazení jsou zobrazeny všechny proběhlé přechody, závislosti mezi nimi jsou vyznačeny šipkami. Kritická cesta je zobrazena červenou barvou. Poloha přechodu vzhledem k ose y je určena podle procesu, na kterém proběhl, poloha vzhledem k ose x závisí na pořadí jednotlivých přechodů. Původně byla poloha vzhledem k ose x určována podle času odpálení přechodu, od toho ale bylo ustoupeno z důvodu překrývání jednotlivých přechodů při nízkých časových intervalech. přechody jsou zobrazeny jako čtverce, kdy čísla v jednotlivých čtvercích představují id přechodu.Přechody patřící na kritickou cestu jsou zobrazeny červeně vyplněnými čtverci, hrany mezi těmito přechody jsou taktéž červené. Legenda přiřazující názvy přechodů k jejich id je umístěna nad vrchním okrajem oblasti grafu.

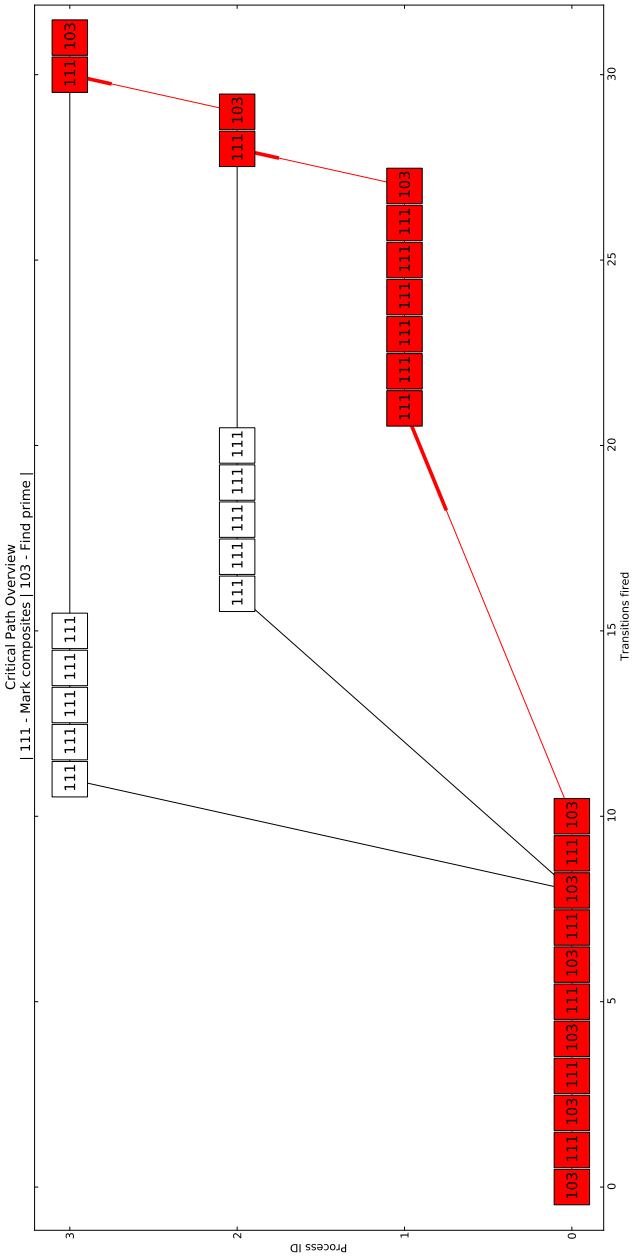
Ukázka Celkového zobrazení příkladu Workers (síť projektu k vidění na obrázku 3) lze nalézt na obrázku 13. Z grafu je vidět, že program využil všech čtyř dostupných procesů, kdy na procesu 0 probíhaly výhradně přechody *divide* a koncový přechod write results, zbylé tři procesy byly programem určeny pro přechody *compute*. Dále lze vyčíst, že cesty nekritické se od cesty kritické významně neliší.

U příkladu projektu Sieve na obrázku 14 je na první pohled zřejmé, že rozložení práce na jednotlivé procesy není ideální. Více než jeden proces naráz je vytížen pouze v krátké části běhu programu a hodnota float u nekritických cest bude poměrně vysoká.

K vytvoření a zobrazení grafu jsou použity knihovny `pyplot` z `matplotlib` API a `networkx`. První se vytvoří `figure`, ve které se graf zobrazí a jako hodnoty na ose y se umístí jednotlivé procesy, se kterými program pracoval. Poté se vytvoří instance třídy



Obrázek 13: Celkový pohled projektu Workers



Obrázek 14: Celkový pohled projektu Sieve

`DiGraph` reprezentující orientovaný graf a připraví se seznamy pro uzly grafu ležící na kritické cestě, pro uzly mimo kritickou cestu a pro popisky jednotlivých uzlů. Také se vytvoří slovník, který bude obsahovat seznam všech uzlů a jejich pozice. Poté následuje průchod obráceným seznamem všech přechodů, kdy se u každého přechodu vytvoří uzel a přidá se do patřičného seznamu. Dále se uzel přidá do slovníku s polohou, která je určena pořadovým číslem přechodu a jeho procesem, a id přechodu se přidá do seznamu popisku. U každého přechodu se vytvoří hrany k přechodům z jeho seznamu závislostí. Po zpracování všech přechodů se jednotlivé hrany rozdělí do dvou seznamů, kdy jeden obsahuje hrany kritické cesty a druhý hrany ostatní. Následně se vykreslí všechny uzly, ty ze seznamu kritické cesty jako červeně vyplněné čtverce, ostatní jako bíle vyplněné. Dále hrany, kdy ty patřící na kritickou cestu jsou zobrazeny jako červené šípky a ostatní jako jednoduchá černá čára. Nakonec se jednotlivým uzlům přidají popisky, vytvoří se legenda grafu, popíší se jednotlivé osy a obrázek s grafem se zobrazí uživateli.

```
def create_critical_path_overview(self):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    yticks = []
    for x in range(0,self.process_count):
        yticks.append(x)
    ax.set_yticks(ticks=yticks)
    G = nx.DiGraph()
    crp, crpNodes, nonCrpNodes = []
    pos = dict()
    labels = {}
    for t in reversed(self.transitions_list):
    temp = 0
    for item in reversed(t.data_dependant_on):
        G.add_edges_from([(item,t.position_in_list)])
        if t.on_critical_path and self.transitions_list[item].on_critical_path and temp == 0:
            crp.append((item, t.position_in_list))
            temp = 1
    pos[t.position_in_list] = (t.position_in_list,t.process_id)
    labels[t.position_in_list] = t.transition.id
    if t.on_critical_path == True:
        crpNodes.append(t.position_in_list)
    else:
        nonCrpNodes.append(t.position_in_list)
    edge_colours = ['black' if not edge in crp else 'red' for edge in G.edges()]
    black_edges = [edge for edge in G.edges() if edge not in crp]
    nx.draw_networkx_nodes(G, pos, node_color='w',nodelist=nonCrpNodes, node_size=1500,
        node_shape='s')
    nx.draw_networkx_nodes(G, pos, node_color='r',nodelist=crpNodes, node_size=1500,
        node_shape='s')
    nx.draw_networkx_edges(G, pos, edgelist=crp, edge_color='r', arrows=True)
    nx.draw_networkx_edges(G, pos, edgelist=black_edges, arrows=False)
    nx.draw_networkx_labels(G, pos, labels, font_size=16)
    title = " Critical␣Path␣Overview\n|␣"
    for t in self.transitions.values():
        title = title + str(t.id) + "␣-␣" + t.name + "␣|␣"
    plt.title(title)
```

```
plt.ylabel("Process ID")
plt.xlabel("Transitions fired")
plt.show()
```

Výpis 3: Vytvoření celkového pohledu

4.4.2.2 Detailní zobrazení (Detail view) Detailní zobrazení na rozdíl od celkového nezobrazuje všechny proběhlé přechody, ale zaměřuje se pouze na kritickou cestu. Proto jsou mezi jednotlivé přechody přidána relevantní místa. Přechody jsou stále zobrazeny jako čtverce, místa jsou zobrazena jako kruhy. Čísla uvnitř jednotlivých obrazců představují id prvku. Ukázka vizualizace Detailního zobrazení příkladu Workers je k nalezení na obrázku 15. Všechny zobrazené přechody patří na kritickou cestu, hrany jsou v tomto zobrazení pouze mezi místem a přechodem, respektive mezi přechodem a místem. Poloha jednotlivých přechodů je závislá na procesu, poloha míst se odvíjí od polohy posledního proběhlého přechodu.

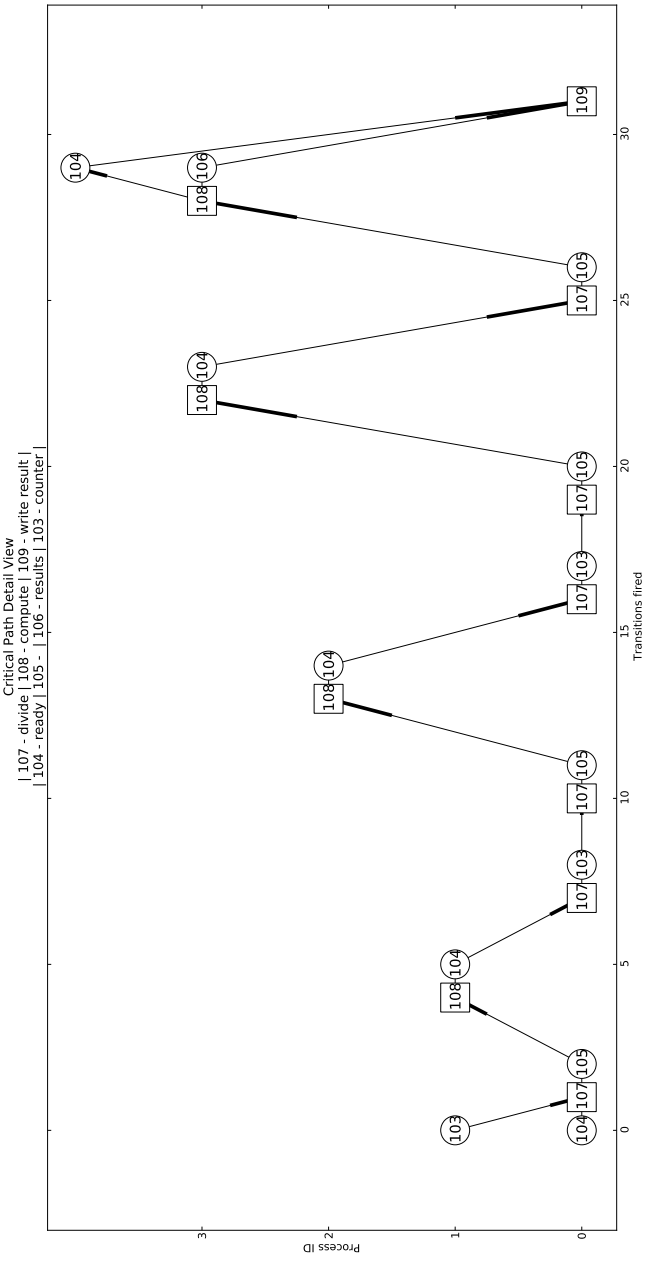
Způsob tvorby grafu je podobný tomu u celkového pohledu s několika rozdíly. Vykreslují se pouze uzly znázorňují přechody na kritické cestě, ke kterým se přidá grafické znázornění míst, které se mezi prvky cesty vyskytují. Kruhy znázorňující místa jsou umístěny bezprostředně za uzel přechodu, který do nich umístil tokeny, pouze u prvního člena kritické cesty jsou zobrazeny místa, ze kterých byly tokeny přechodem odstraněny, před samotný uzel. Barevné odlišení od ostatních prvků už není v tomto zobrazení potřebné, proto jsou všechny hrany a uzly zobrazovány v černé barvě.

4.4.2.3 Rozšíření ovládacích prvků zobrazení V rámci této práce byla zobrazení využívající `matplotlib` rozšířena o ovládací prvky zlepšující orientaci a práci ve zvizualizovaném modelu. Tyto prvky jsou propojeny s funkcemi knihovny `matplotlib` jako události. Rozšíření jsou zpracována ve třídě `GraphAddons` a jsou rozdělena do těchto částí:

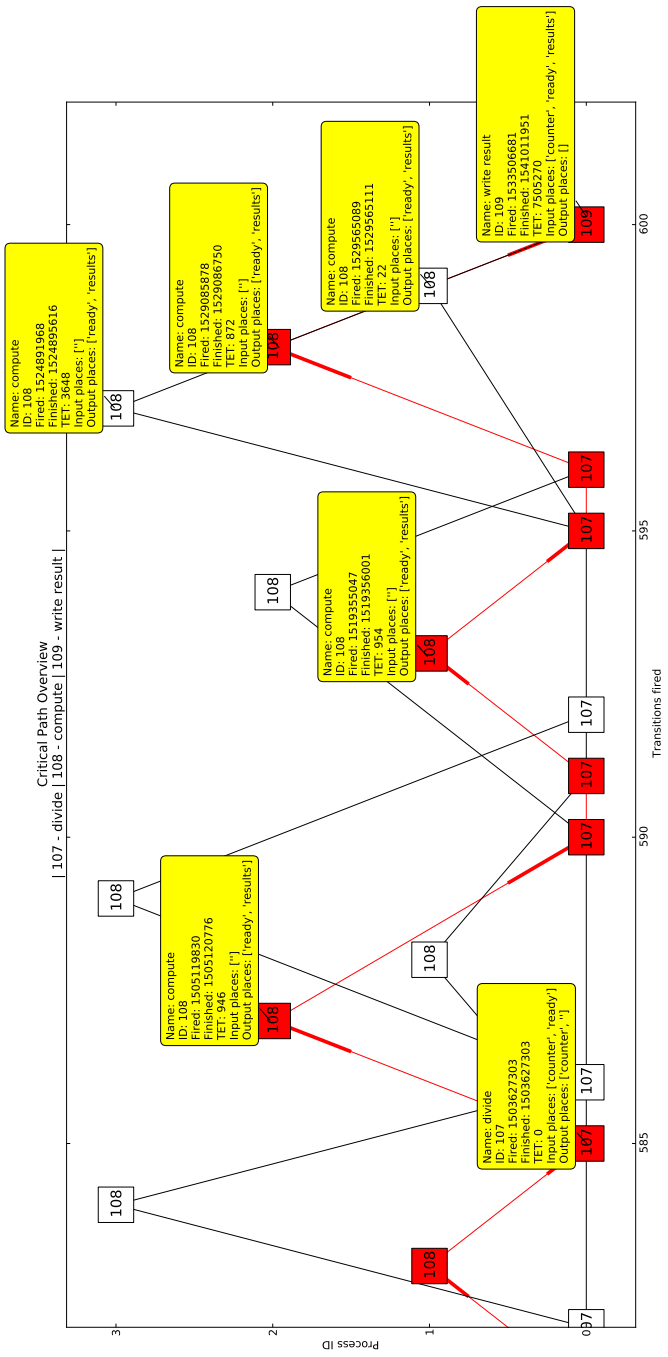
- *zoom_addon* - zajišťuje přibližování a oddalování ve výsledném grafu pomocí kolečka myši
- *pan_addon* - umožňuje použit kolečko myši pro pohyb v grafu
- *keys_addon* - mapuje rozšířené ovládací prvky na vybrané klávesy
- *annote_addon* - zprostředkovává zobrazení informací o přechodech, použito pouze v Celkovém zobrazení

Tato rozšíření umožňují efektivnější orientaci a přizpůsobení vlastních grafu. K dispozici je možnost zamykaní jednotlivých os, přednastavené hodnoty přiblížení, skokové posouvaní se grafem a zobrazení informací o jednotlivých přechodech. Detailní popis přidaných ovládacích prvků je k dispozici v příloze B. V ukázce na obrázku 16 je k vidění část grafu projektu Workers s několika zobrazenými anotacemi.

Cílem těchto ovládacích prvků je zejména zlepšení práce s grafy obsahující stovky až tisíce přechodů.



Obrázek 15: Detailní pohled projektu Workers



Obrázek 16: Část grafu projektu Workers se zobrazenými anotacemi

CP	ID	Name	Process	Fired time	Finished time	TET
True	103	Find prime	0	11793557	11885912	92355
True	111	Mark composites	0	11896901	11898514	1613
True	103	Find prime	0	11899349	11946667	47318
True	111	Mark composites	0	11952011	11952924	913
True	103	Find prime	0	11953631	11997469	43838
True	111	Mark composites	0	12002489	12003227	738
True	103	Find prime	0	12003928	12047218	43290
True	111	Mark composites	0	12052070	12052704	634
True	103	Find prime	0	12053400	12096818	43418
True	111	Mark composites	3	19834535	19835608	1073
True	111	Mark composites	3	19836336	19837173	837
True	111	Mark composites	3	19837884	19838483	599
True	111	Mark composites	3	19839162	19839754	592
True	111	Mark composites	3	19840449	19841011	562
True	111	Mark composites	3	46548627	46549780	1153
True	103	Find prime	3	46550890	46813930	263040

CP	ID	Name	Process	Fired time	Finished time	TET
True	103	Find prime	0	11793557	11885912	92355
True	111	Mark composites	0	11896901	11898514	1613
True	103	Find prime	0	11899349	11946667	47318
True	111	Mark composites	0	11952011	11952924	913
True	103	Find prime	0	11953631	11997469	43838
True	111	Mark composites	0	12002489	12003227	738
True	103	Find prime	0	12003928	12047218	43290
True	111	Mark composites	0	12052070	12052704	634
True	103	Find prime	0	12053400	12096818	43418
False	111	Mark composites	0	12101381	12101949	568
False	103	Find prime	0	12102643	12260588	157945
False	111	Mark composites	1	14887347	14888879	1532
False	111	Mark composites	1	14889712	14890620	908
False	111	Mark composites	1	14891356	14892022	666
False	111	Mark composites	1	14892695	14893292	597
False	111	Mark composites	1	14893966	14894452	486
False	111	Mark composites	1	14895126	14895654	528

ID	Process	Fired time
103	0	11793557
111	0	11896901
103	0	11899349
111	0	11952011
103	0	11953631
111	0	12002489
103	0	12003928
111	0	12052070
103	0	12053400
111	3	19834535
111	3	19836336
111	3	19837884
111	3	19839162
111	3	19840449
111	3	46548627
103	3	46550890

Obrázek 17: Kritická cesta příkladu Sieve zobrazená tabulkami Critical Path only, All transitions a Critical Path export

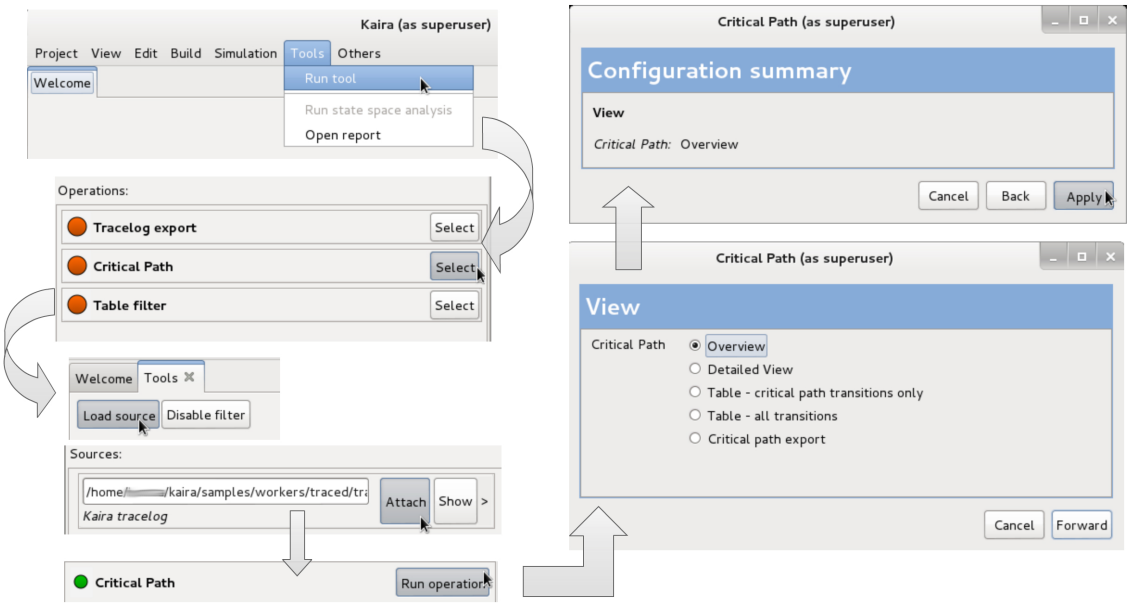
4.4.2.4 Zobrazení pomocí tabulek Rozšíření umožňuje vygenerovat reprezentaci nalezené kritické cesty pomoci CSV (comma-separated value) tabulky, k čemuž je využito již existující součásti Kairy pro práci s CSV tabulkami. K dispozici jsou tři varianty těchto tabulek. Varianty *Critical Path only*, obsahující pouze přechody kritické cesty, a *All transitions*, obsahující všechny přechody, jsou tvořeny těmito informacemi:

- *CP* - True v případě, že přechod patří na kritickou cestu, False v případě opačném
- *ID* - id přechodu
- *Name* - jméno přechodu
- *Process* - proces, na kterém přechod proběhl
- *Fired time* - čas odpálení přechodu
- *Finished time* - čas odpálení přechodu
- *TET* - doba, kterou přechod potřeboval k provedení

Varianta *Critical Path export* obsahuje pouze informace potřebné k identifikaci přechodů tvořící kritickou cestu, tedy:

- *ID* - id přechodu
- *Process* - proces, na kterém přechod proběhl
- *Fired* - čas odpálení přechodu

Výsledná tabulka je uživateli poskytnuta k zobrazení přímo v prostředí Kairy. Příklad tabulkek získaných z příkladu Sieve na obrázku 17.



Obrázek 18: Návod ke spuštění rozšíření nástroje Kaira pro vizualizaci kritické cesty

4.5 Použití vizualizace

Po spuštění programu Kaira najdeme rozšíření umožňující vizualizaci kritické cesty mezi ostatními rozšířeními v nabídce *Tools* pod názvem *Critical Path*. Po zvolení tohoto rozšíření je třeba připojit správně vytvořený tracelog. Ten vybereme po stisknutí *Load Source* a připojíme jej stisknutím tlačítka *Attach*. Poté můžeme spustit rozšíření stiskem tlačítka *Run*, v zobrazivší se nabídce si vybereme způsob zobrazení, stiskem Forward se přesuneme na obrazovku s potvrzením. Po stisku tlačítka *Apply* se zobrazí zvolená vizualizace ve formě interaktivního obrázku, popř. se v nabídce *Sources* objeví jako nový zdroj vygenerovaná tabulka, která lze zobrazit stiskem tlačítka *Show*. Graficky je postup znázorněn na obrázku 18.

Jako výsledek se uživateli zobrazí interaktivní obrázek, se kterým může dále pracovat. Uživatel může graf libovolně přibližovat a oddalovat, pohybovat s ním v rámci zobrazení a upravovat velikost okrajů a prostoru pro graf. Výslednou podobu grafu je možno uložit v následujících formátech:

- Portable Network Graphics (*.png)
- Enhanced Metafile (*.emf)
- Encapsulated Postscript (*.eps)
- Joint Photographic Experts Group (*.jpeg, *.jpg)
- Portable Document Format (*.pdf)

- Postscript (*.ps)
- Raw RGBA bitmap (*.raw, *.rgba)
- Scalable Vector Graphics (*.svg, *.svgz)
- Tagged Image File Format (*.tif, *.tiff)

5 Budoucí práce

Jednou z variant budoucího rozšíření práce je zlepšení problémových části současného způsobu zobrazení, zejména problémů s rychlosti aktualizace a překreslení u grafu s vysokým množstvím přechodů. Také by se uživateli mohly poskytnout větší možnosti ovlivnění konečné podoby vytvořených grafů, např. výběrem barevného schéma či možnosti schovávat určité prvky grafu. Také se nabízí možnost rozšířit detailní zobrazení.

Další možností budoucího vývoje je lepší zapracování tohoto rozšíření do prostředí Kairy, např. mezi vložení výsledných grafu mezi grafy dostupné při základním zobrazení tracelogu. Této integraci ale musí předcházet vyřešení problému s nepřehledností napevno vygenerovaných grafů.

Větším projektem by bylo vytvoření interaktivní mapy jak bylo nastíněno v 4.4.1. Její funkcionalita by se neomezovala pouze na zobrazení kritické cesty, ale byla by komplexním nástrojem pro analýzu proběhlých programů.

6 Závěr

Úkolem práce bylo rozšířit nástroj Kaira o vyhledání a vizualizaci kritické cesty z tracelogů vygenerovaných aplikací. Před započetím práce jsem se nejprve seznámil s nástrojem Kaira, jeho prostředím, tvorbou programů, simulacemi a zejména s možnostmi a formátem zaznamenávání dat získaných při běhu programu. V získaných datech jsem identifikoval data potřebná ke splnění zadaného úkolu. Po načtení byla tato data zpracována a byla v nich nalezena kritická cesta. Ta se uživateli zobrazuje v jedné ze dvou možností podle jeho výběru jako interaktivní obrázek, nebo jsou informace o přechodech vygenerovány do tabulky.

Při vybírání možnosti vizualizace se rozhodnutí několikrát změnilo. Ačkoliv základní představa o výsledku se měnila minimálně, přesný způsob provedení se dlouho nedařilo zvolit. Nakonec bylo zvoleno zobrazení ve formě orientovaného grafu s přechody jako uzly a vyznačenými závislostmi. Později byla přidána možnost zobrazení detailu kritické cesty se zobrazenými místy a jako poslední byla přidána možnost zobrazení základních informací o prvcích kritické cesty ve formě tabulky.

Vzhledem k velké rozmanitosti programů. které jdou v Kaiře vymodelovat, nebylo možné vymyslet univerzální způsob zobrazení. Při snaze o vytvoření obrázku vizualizace docházelo zejména u programu s velkým počtem provedených přechodu k překrývání uzlů grafu. Proto jsem jako formu zobrazeni zvolil interaktivní obrázek, který s implementovaným rozšířeným ovládáním umožňuje pracovat i s grafy s velkým množstvím přechodů.

7 Reference

[1] PETRI, Carl a Wolfgang REISIG. *Petri net.* Scholarpedia. 2008, vol. 3, issue 4, s. 6477-. DOI: 10.4249/scholarpedia.6477. Dostupné z: http://www.scholarpedia.org/article/Petri_net

[2] BÖHM, Stanislav a Marek BĚHÁLEK. *Usage of petri nets for high performance computing. Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing - FHPC '12.* New York, New York, USA: ACM Press, 2012, s. 37-. DOI: 10.1145/2364474.2364481. Dostupné z: http://dl.acm.org/citation.cfm?doid=2364474.2364481

[3] JENSEN, K. *Coloured petri nets: basic concepts, analysis methods, and practical use.* 2nd ed. New York: Springer, 1997-, v. <1 >. ISBN 35406094311-.

[4] BÖHM, Stanislav *Unifying Framework For Development of Message-Passing Applications.* Ostrava: 2013. Ph.D. Thesis. Faculty of Electrical Engineering and Computer Science VŠB – Technical University of Ostrava.

[5] KELLEY, James E. a Morgan R. WALKER. *Critical-path planning and scheduling. Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '59 (Eastern)* New York, New York, USA: ACM Press, 1959, s. 160-173. DOI: 10.1145/1460299.1460318. Dostupné z: http://portal.acm.org/citation.cfm?doid=1460299.1460318

[6] ARMSTRONG-WRIGHT, A. *Critical path method: introduction and practice.* Harlow: Longmans, 1969, xi, 113 p. ISBN 05-824-1040-1.

[7] GEIMER, Markus, Felix WOLF, Brian J. N. WYLIE, Erika ÁBRAHÁM, Daniel BECKER a Bernd MOHR. *The Scalasca performance toolset architecture.* Concurrency and Computation: Practice and Experience. 2010, n/a-n/a. DOI: 10.1002/cpe.1556. Dostupné z: http://doi.wiley.com/10.1002/cpe.1556

[8] BOHME, David, Felix WOLF, Bronis R. DE SUPINSKI, Martin SCHULZ a Markus GEIMER. *Scalable Critical-Path Based Performance Analysis.* 2012 IEEE 26th International Parallel and Distributed Processing Symposium. IEEE, 2012, s. 1330-1340. DOI: 10.1109/IPDPS.2012.120. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6267934

[9] DOOLEY, Isaac a Laxmikant V. KALE. *Detecting and using critical paths at runtime in message driven parallel programs.* 2010 IEEE International Symposium on Parallel. IEEE, 2010, s. 1-8. DOI: 10.1109/IPDPSW.2010.5470844. Dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5470844

A Obsah CD

Přiložené CD obsahuje tyto složky:

- ExtensionOnly - složka obsahující soubor critpath.py
- Kaira - složka s nástrojem Kaira. Soubor critpath.py vytvořený v rámci této práce je umístěn ve složce \kaira\gui\extensions\

Dále je přiložen soubor readme.txt s instalačními pokyny a seznamem ovládacích prvků rozšíření.

B Seznam ovládacích prvků vygenerovaných grafů

Kromě původní lišty nástroju figury (zleva doprava: reset grafu, předchozí stav, následující stav, posun, lupa, nastavení okna, uložit) byly vygenerované grafy rozšířeny o následující prvky:

B.1 Pohyb v grafu

- Kolečko myši - umožňuje přibližování/oddalování pomocí točení kolečka k/od polohy kurzoru myši. Po stisknutí kolečka je možno grafem posouvat.
- Klávesa **X** - zamyká/odemyká možnost měnění osy X
- Klávesa **Y** - zamyká/odemyká možnost měnění osy Y
- Klávesa **Home** - vrátí osy X a Y do stavu při prvním zobrazení grafu
- Klávesa **PageUp** - zdvojnásobí počet zobrazených procesů
- Klávesa **PageUp** - zmenší počet zobrazených procesů na polovinu
- Klávesa ↑ - posun nahoru po ose Y
- Klávesa ↓ - posun dolů po ose Y
- Klávesa ← - posun doleva po ose X
- Klávesa → - posun doprava po ose X
- Klávesa **+** - zvětšuje velikost posunu
- Klávesa **-** - zmenšuje velikost posunu
- Klávesa ***** - vrací velikost posunu na původní hodnotu
- Klávesa **1** - nastaví měřítko osy X na 10 přechodů
- Klávesa **2** - nastaví měřítko osy X na 20 přechodů
- Klávesa **3** - nastaví měřítko osy X na 50 přechodů
- Klávesa **4** - nastaví měřítko osy X na 100 přechodů
- Klávesa **5** - nastaví měřítko osy X na 200 přechodů
- Klávesa **6** - nastaví měřítko osy X na 500 přechodů
- Klávesa **7** - nastaví měřítko osy X na 1000 přechodů

B.2 Zobrazení informací v grafu

- Levé tlačítko myši - kliknutím na přechod zobrazí dočasnou anotaci s informacemi
- Pravé tlačítko myši - kliknutím na přechod zobrazí trvalou anotaci s informacemi
- Klávesa **Backspace** - maže trvalé anotace od poslední přidané
- Klávesa **H** - zobrazí informace o stavu zamknutí os X a Y a aktuální velikosti posunu